Artificial Intelligence Laboratory
Massachusetts Institute of Technology

AIM 429

June 1977

A PROOF-CHECKER FOR DYNAMIC LOGIC

S.D. Litvintchouk and V.R. Pratt
Massachusetts Institute of Technology
Cambridge, MA 02139
June 20, 1977

Abstract

We consider the problem of getting a computer to follow reasoning conducted in dynamic logic. This is a recently developed logic of programs that subsumes most existing first-order logics of programs that manipulate their environment, including Floyd's and Hoare's logics of partial correctness and Manna and Waldinger's logic of total correctness. Dynamic logic is more closely related to classical first-order logic than any other proposed logic of programs. This simplifies the design of a proof-checker for dynamic logic. Work in progress on the implementation of such a program is reported on, and an example machine-checked proof is exhibited.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

A PROOF-CHECKER FOR DYNAMIC LOGIC

S.D. Litvintchouk and V.R. Pratt
Massachusetts Institute of Technology
Cambridge, MA 02139

Introduction

The logical language.

Our objective is to be able to discuss programs with a computer. The prerequisites are a language for holding the conversation in, and reliable criteria for following a line of reasoning expressed in this language. We adopt a simple language having just four basic constructs. Three of these constructs come from ordinary logic; they are function symbols, predicate symbols, and logical connectives. (We lump constants and variables together with the zeroary function symbols.) The fourth construct, while not a familiar one in logic, is nevertheless one that occurs in everyday conversations about programs; it is the notion of "after executing program α." For example we may say in ordinary conversation, "After executing the program X:=1, X is equal to 1."

While these four constructs may not seem very much to go on, they are in fact sufficient for almost any "first-order" conversation about the input-output behavior of programs. They may express such diverse concepts as partial correctness, termination, equivalence, determinism versus nondeterminism, totality, reversibility of a process, accessibility of states, weakest antecedents, strongest consequents, weakest and strongest invariants, and convergents. They also shed new light on the axioms relevant to quantifiers in first-order predicate calculus by treating them from the programmer's point of view rather than the logician's.

We abbreviate "after executing program α" to $[\alpha]$, so that the observation of the first paragraph condenses to $[X:=1]X=1$. (We have found it convenient in conversation to pronounce $[\alpha]$ as "box α.") We shall later find useful the dual concept $\neg[\alpha]\neg$ which we write $\langle\alpha\rangle$, pronounced "diamond α." The notation is borrowed from modal logic. Dynamic logic is more intimately connected with modal logic than one might at first suppose; the connection is discussed in more detail in section 3.2 of [21]. Fischer and Ladner [6] demonstrate the

connection between various restrictions of dynamic logic and the classical systems K, T, S4 and S5 of modal logic. We call $[\alpha]$ and $\langle\alpha\rangle$ modalities (respectively box and diamond modalities), and formulae of the form $[\alpha]P$ and $\langle\alpha\rangle P$ modal formulae. We shall call a quantifier-free logic augmented with such modalities a dynamic logic. Syntactically, modalities behave exactly like logical negation; they are placed in front of a formula, and their precedence is such that $[\alpha]P\wedge Q$ is parsed $([\alpha]P)\wedge Q$, just as $\neg P\wedge Q$ would have been parsed $(\neg P)\wedge Q$.

The programming language

In order to understand the meaning of a formula such as $\neg[X:=1]$false, we first need a precise account of X:=1. We shall think of programs solely in terms of their effect on the state of the world. A state is defined by the values taken on by the function and predicate symbols of the language in some domain. (A logician would call a state an interpretation.) We call the set of all possible states (keeping the domain fixed) the universe. Thus a universe is defined by the available function and predicate symbols and the choice of domain.

We could restrict our attention to deterministic programs, permitting us to think of them as functions from states to states. As we shall see later however, reasoning nondeterministically about deterministic programs can simplify the argument. Hence we shall allow for nondeterministic programs by capturing the effect of a program on a universe as a binary relation on that universe. This of course means that we will be able to discuss nondeterministic programs in general. However, the question of what first-order facts one wants to assert about nondeterministic programs is presently the subject of much discussion in the literature (see [5] in particular), and we shall avoid that issue in this paper beyond observing that dynamic logic as presented here can express many useful ideas about nondeterministic programs.

In treating programs as binary relations we shall make use of the usual notation that $\mathcal{I}\alpha\mathcal{J}$ is true just when $\mathcal{I}$ and $\mathcal{J}$ are related by α. It is convenient to identify the relation α as its graph, the set of pairs of states related by α.

Programming constructs

The programs we want to discuss have five constructs. These constructs, while not all entirely conventional, have been chosen primarily for the ease with which one can discuss programs written using them.

(i) Assignments. X:=1 is an instance of an assignment, as is C(I,K):=C(I,K)+A(I,J)×B(J,K). In general an assignment is a pair of terms (respectively the left-hand and right-hand sides of the

assignment) of our logical language. (A term is an expression constructed solely from function symbols.) We shall take zeroary function symbols to be ordinary variables. Then when the left-hand term is simply a zeroary function symbol the assignment is simple variable assignment; for other left-hand sides we have array assignments. Formally, the simple variable assignment X:=t, where t is an arbitrary term, is $\{(\mathcal{I},\mathcal{J})|X_{\mathcal{J}}=t_{\mathcal{J}}$ and otherwise $\mathcal{I}=\mathcal{J}\}$. ($X_{\mathcal{J}}$ denotes the value of X in $\mathcal{J}$, $t_{\mathcal{J}}$ the value of t in $\mathcal{I}$.) Array assignments are slightly harder to define; see [21].

(ii) Tests. Conditionals are usually introduced with "if-then-else." However the rules of reasoning (our axiom system) can be simplified by using a "smaller" notion of conditional, the test, which we shall use in conjunction with the next two constructs to synthesize if-then-else. X>0? is an instance of a test, as is J=0∨Pattern(J)=Text(K)?. In general a test P? is constructed from a formula P of the logical language. The idea of a test is that a computation may proceed past a test just when that test evaluates to <u>true</u> in the current environment, otherwise the computation must block (which for our purposes is equivalent to going into an infinite loop). Formally, the test P? is the restriction of the identity binary relation on the universe to those states satisfying P, i.e. $\{(\mathcal{I},\mathcal{I})|\mathcal{I}\models P\}$. Most of what we say holds even for tests containing modalities, corresponding to the side-effect-free programming construct "if P <u>would be</u> the result of running α then..." However we shall confine our examples to the more familiar modality-free variety.

(iii) Alternation. Execution of $\alpha|\beta$ means the execution of either one (but not both) of α and β, the choice being made nondeterministically. Formally, the relation $\alpha|\beta$ is the union of the relations α and β.

(iv) Sequencing. Execution of $\alpha;\beta$ means execution of first α then β. Formally, $\alpha;\beta$ is the composition of α and β.

Using tests, alternation, and sequencing, we may express "if P then α else β," where P is a formula and α and β are programs, as $P?;\alpha|\neg P?;\beta$. In effect, P and $\neg P$ act as "guards," to use Dijkstra's [5] terminology. $P?;\alpha$ can only be executed when P holds, and conversely for $\neg P?;\beta$. Hence when P is true $P?;\alpha|\neg P?;\beta$ must be equivalent to α, and otherwise to β, which is the property "if P then α else β" should have.

(v) Iteration. Execution of α^* means executing α zero or more times, the number of times being chosen nondeterministically. Formally, α^* is the reflexive transitive closure of α.

Using tests, sequencing, and iteration, we may express "while P do α" in much the same spirit as if-then-else, namely as $(P;\alpha)^*;\neg P$. This permits α to be iterated for as long as P

remains true. Moreover, the iteration may not terminate while P remains true, on account of the ¬P guarding the exit. (This usage of a guard at the end is not permitted in [5].)

With these five constructs we can express any flowchart program that has decision boxes and manipulates arrays. This can be done without introducing additional variables. This follows from the fact that state transition diagrams can always be translated into equivalent regular expressions. This is not possible with assignments, sequencing, if-then-else and while-do [2], the difference having to do with the determinism of the latter.

Quasi-programming constructs

In addition to the five constructs for our programming language, we shall find two more constructs of interest, not in writing programs but in talking about them.

(vi) Random assignment. $X:=?$ is an instance of a random assignment, which nondeterministically assigns an arbitrary element of the domain to X. Consider the sense of $[X:=?](X<0 \vee X \geqslant 0)$. This says that no matter what element is assigned to X, after the assignment X will be either negative or non-negative. This captures what is meant by $\forall X(X<0 \vee X \geqslant 0)$. This demonstrates that we can introduce the ordinary quantifier $\forall X$ into dynamic logic as just another modality $[X:=?]$. Though we shall adhere to the standard notations $\forall X$ and $\exists X$ it should be understood that these stand respectively for $[X:=?]$ and $\langle X:=? \rangle$.

(vii) Converse. Execution of α^- means the reverse execution of α. Formally, α^- is the converse of α, satisfying $\mathcal{I}\alpha\mathcal{J} \equiv \mathcal{J}\alpha^-\mathcal{I}$. This permits us to reason either forwards or backwards about a program's behavior. We mean this in the sense that (i) $[\alpha]P$ is a claim made before execution of α based on the claim P that is supposed to hold after execution of α (backward reasoning), and (ii) $[\alpha^-]P$ is a claim made after the execution of α based on the claim P that is supposed to hold before execution of α (forward reasoning). We have not capitalized on converse as much as we would like in our work to date.

Truth-value Semantics of Dynamic logic

Now that we have settled on the programming language, we can return to the question of what $\neg[X:=1]$false means, or more generally what any formula containing $[\alpha]P$ means. It is important here to realize the distinction between truth and validity. What we are about to define is the truth value of a formula of dynamic logic in a single state. This is to be contrasted with, say, Hoare's notion of "$P\{\alpha\}Q$," whose truth is not defined on a state-by-state basis but rather is defined for the whole universe, and so corresponds to the usual notion in

logic of validity.

In state $\mathcal{I}$, $[\alpha]P$ is true just when P is true in every state $\mathcal{J}$ satisfying $\mathcal{I}\alpha\mathcal{J}$. That is, P is true no matter which state α terminates in when started in state $\mathcal{I}$. It follows that $\langle\alpha\rangle P$ is true just when P is true in some state $\mathcal{J}$ satisfying $\mathcal{I}\alpha\mathcal{J}$, that is, when it is possible for α to terminate and satisfy P if started in state $\mathcal{I}$.

Expressive power of dynamic logic

We may now show how dynamic logic may be used to express a variety of concepts.

$P\{\alpha\}Q$	$\models(P\supset[\alpha]Q)$
Termination analogue of $P\{\alpha\}Q$	$\models(P\supset\langle\alpha\rangle Q)$
$\alpha\equiv\beta$	$\models\forall X((\langle\alpha\rangle Y=X)\equiv(\langle\beta\rangle Z=X))$

(This assumes that Y,Z are the respective output variables of α,β. While this generalizes to programs with any finite set of output variables, it does not generalize to programs with arrays as output unless we introduce second order quantifiers.)

α deterministic	$\models\forall X(\langle\alpha\rangle Y=X\supset[\alpha]Y=X)$

(As for equivalence, Y is the output variable of α.)

α always halts	$\models\langle\alpha\rangle$true
α 1-1	$\models\forall X(\langle\alpha^-\rangle Y=X\supset[\alpha^-]Y=X)$
α onto	$\models\langle\alpha^-\rangle$true
α halts in this state	$\langle\alpha\rangle$true
weakest antecedent	$[\alpha]P$
strongest consequent	$\langle\alpha^-\rangle P$
weakest invariant	$[\alpha^*]P$ (see [9] for proof)
strongest invariant	$\langle\alpha^-{}^*\rangle P$ " " " "
convergent	$\langle\alpha^N\rangle P$ " " " "

The reader wishing to pursue these concepts further is referred to [9]. Some simple statements expressible in dynamic logic that do not fall into any of the above categories, and are not expressible in Hoare's partial correctness formalism or the total correctness formalism of Manna and Waldinger [17], are:

"If you set Y to X+5 and then [Y:=X+5;Y:=Y+2*]

add 2 to Y an indefinite number of times then it is possible by repeatedly executing Y:=Y-1 to make Y=X"	$\langle Y:=Y-1^{*}\rangle Y=X$
"If P holds then after executing α we will be in a state accessible via α from some state satisfying P."	$P \supset [\alpha]\langle\alpha^{-}\rangle P$

All of the above concepts can be stated in a second order logic that permits explicit manipulation of states and/or programs as individuals, as in [3] where states can be quantified over, or [10] where programs are terms. The interest in dynamic logic is that it achieves its expressive power using only first-order language. The advantage of keeping the language restricted in this way is that it is easier to completely axiomatize parts of the logic, though loops present an insurmountable obstacle to completeness as demonstrated in Theorem 16 of [21].

An axiom system for dynamic logic

Before axiomatizing the programming language, let us begin with a sound complete axiom system for first-order logic. A novelty of this system is that it separates into logical and non-logical components what are usually taken to be entirely logical rules and axioms, on the principle that facts about X:=? are program-specific facts. This permits a programmer to apply his intuition about programs to the problem of understanding the significance of each axiom.

Logical Axioms

All tautologies of Propositional Calculus.

$[\alpha](P \supset Q) \supset ([\alpha]P \supset [\alpha]Q)$. (Axiom M)

Logical Inference Rules

$P, P \supset Q \vdash Q$. (Modus Ponens)

$P \vdash [\alpha]P$ (Necessitation; subsumes generalization, $P \vdash \forall x P$).

Non-logical Axioms

$\forall X P \supset P^{T}_{X}$ (any term T; P^{T}_{X} is P with T for X)

$P \supset \forall X P$ unless X occurs free in P

Axiom M can be thought of as a claim about programs; it says that for all states ϑ, if P

implies Q in every state $\mathfrak{J}$ that can be reached from $\mathfrak{I}$ by executing α, and if P holds in every state $\mathfrak{J}$ similarly accessible from $\mathfrak{I}$ via α, then Q holds in every state $\mathfrak{J}$ accessible from $\mathfrak{I}$ via α.

The second inference rule (the rule of necessitation of modal logic) can be considered as an upper bound on the power of programs, which cannot falsify theorems. If P is a theorem then P is true in every state, including states accessible via α.

In our system it is straightforward to prove as theorems the axioms of, say, Mendelson's system K [18] (p. 57), and it should be clear that the second rule subsumes the rule of generalization; in fact, if the only modalities allowed are those with values of the form X:=? then the rule of necessitation is the rule of generalization, and the theorems of this system coincide with the theorems of K. It is interesting to note that Mendelson manages to express as one axiom what we take two to express, namely our Axiom M and the second quantification axiom. The advantage of our decomposition of this axiom is that we get two axioms about quantifiers that serve respectively as a lower and an upper bound on what the binary relation X:=? may be.

So far we have only given axioms for random assignments. Now let us axiomatize the four loop-free programming constructs.

$[P?]Q \equiv P \supset Q$

$[X:=t]P \equiv P^t_X$ (see [21] for array assignment)

$[\alpha|\beta]P \equiv [\alpha]P \wedge [\beta]P$

$[\alpha;\beta]P \equiv [\alpha][\beta]P$

Interestingly (but fairly obviously, as demonstrated in [21]), the axiom system with these four new axioms remains sound, complete and effective. (It is possible to give further axioms to handle the converse operation, still preserving soundness, completeness and effectiveness. However we shall not make use of this in the following.)

A derived rule

We could at this point proceed with the discussion of our ultimate objective, the construction of a proof-checking program that would check proofs expressed in the above axiom system. Unfortunately the above system is too weak to permit reasonably succinct proofs; for example, it appears that 6 lines are needed to prove $\langle X:=1\rangle X=1$ from the assumption 1=1 using the above system. In this section we explore a derived rule with an eye on strengthening the axioms and rules. In this respect we are emulating J.A. Robinson [22], who prescribed a new rule to

facilitate the construction of automatic theorem-provers. The constraints on a proof-checker are somewhat different from those of a theorem-prover, and the arguments for Robinson's resolution rule are not sufficiently compelling for us. In particular, the convenience of having a clause as the unit of information, which helps an automatic theorem prover organize the proof, may be more hindrance than help in a proof-checker because the user may not have conceived his proof in terms of clauses that are disjunctions of literals. This is not to say that we shall not make use of unification; indeed, unification is a most valuable tool in automated logic.

We now give the details of the rule, which we call the Show Rule for lack of a more descriptive term. A proof step using it looks like

Show S {ps} using T0 {p0}, T1 {p1}, T2 {p2}, ...

For the moment ignore the items inside braces { }. Ideally, we would like this rule to apply whenever the formulae T0, T1, T2, ... logically entail the formula S, a semantic characterization of the rule. Unfortunately that would lead to a non-effective proof-checker, since logical entailment is not even partially decidable for our language [9]. Instead we resort to an effective syntactic characterization. This is where the items in braces enter the picture. The braces enclose "templates" which contain the propositional content of the proof step, in the sense that each template is a propositional "approximation" to the formula it follows. For example, we might say

Show [X:=1|X:=2]X>0 {p∧q} using 1>0 {p}, 2>0 {q}.

The template p∧q refers to the result of expanding [X:=1|X:=2]X>0 first to [X:=1]X>0∧[X:=2]X>0 and then to 1>0∧2>0. It should be clear that the two uses of p in the templates refer to the same formula, 1>0, and similarly for the two uses of q. More generally, we shall require only that multiple occurrences of the same letter refer to unifiable formulae.

We check this proof step in two phases, which can be done independently and in either order (or in parallel by two processors). One phase, called IDENTIFY, is to check that repetitions of the same letter can be justified. We do this by attempting to unify corresponding formulae. The other phase, called VERIFY, is to see whether the templates alone constitute a sound argument in modal propositional logic. In this example all modalities were eliminated so that we were left with the argument

Show p∧q using p, q

which is in fact a sound argument of non-modal propositional logic. A situation where modal logic would help is:

Show $[U;V]Y>0\ \{[\alpha][\beta]p\}$
using $[U]X=1\ \{[\alpha]q\}$, $X=1 \supset [V]Y>0\ \{q \supset [\beta]p\}$.

Here we are dealing with "uninterpreted" programs U and V, a situation that arises when we are given a program about which we have previously proved some useful properties and whose code we no longer wish to be bothered with. (This situation arises frequently in the extended example of the next section but one.) In this case, knowing nothing about the programs U and V beyond the facts given, we could not expand them in the way we did with $[X:=1]$, so they carry over to the templates. Here the argument of modal logic is:

Show $[\alpha][\beta]p$ using $[\alpha]q$, $q \supset [\beta]p$.

This argument can readily be seen to follow if we apply Necessitation to $q \supset [\beta]p$ to get $[\alpha](q \supset [\beta]p)$ and hence $[\alpha]q \supset [\alpha][\beta]p$. The rest is propositional reasoning.

The IDENTIFY phase begins by determining what subformula each occurrence of a template letter refers to. This is done by systematically expanding the formula associated with the template containing the given letter until the formula can be matched to the template. Thus $[U;V][W]X=0$ will match $[\alpha][\beta]p$ directly with a matched to U;V, b to W and p to X=0. However $[U;V]X=0$ will not match $[\alpha][\beta]p$ directly but must first be expanded as $[U][V]X=0$. $[V|W]X=0$ will not match $p \wedge q$ directly but must first be expanded as $[V]X=0 \wedge [W]X=0$. Once the formula matches the template, the subformula corresponding to each letter can immediately be determined. Then all the subformulae corresponding to occurrences of the same letter are checked for whether they can be unified. This may require further expansion; for example, attempting to unify $[X:=1]X>0$ and $W>0$ involves eliminating the assignment modality to give $1>0$, and instantiating W as 1, this latter step being performed by a unification algorithm. All instantiations necessary must be compatible with each other.

Any formulae that fail to unify are put to one side while the remainder of the proof step is checked. When that is done, then the failed pairs are expressed as an equivalence and tested by a routine that checks for validity of quantifier-free Presburger arithmetic, in the hope that the formulae turn out to be equivalent on arithmetic grounds. (This together with the Rule of Convergence described in the next section is the only concession to domain-dependencies in the system.)

The VERIFY phase is a satisfiability tester for modal propositional logic. It begins by determining what applications of the Rule of Necessitation are sufficient to make the proof go through. Boxes are then eliminated from the formula by the appropriate generalization of the transformation $\langle\alpha\rangle P \wedge [\alpha]Q \rightarrow \langle\alpha\rangle(P \wedge Q)$, which preserves satisfiability for the intuitively obvious reason that $[\alpha]Q$ acts only as a constraint on those worlds one might construct (in attempting to satisfy $\langle\alpha\rangle P$) that are accessible via a and satisfy P, namely that in any such world Q must be true. In our present implementation, we first eliminate all top-level propositional letters by expressing the formula in conjunctive normal form and applying the Davis-Putnam algorithm for each of those letters. Then we convert the resulting formula involving only modalities to disjunctive normal form and apply the above transformation. Then the process is repeated on the arguments of the top-level diamond modalities. Though this approach can be inefficient, in practice on the kinds of formulae we encounter it is the most efficient of the methods we have tried. With all boxes eliminated, the satisfiability of the result no longer depends on the names of the diamonds; that is, $\langle\alpha\rangle P \vee \langle\beta\rangle Q$ and $\langle\alpha\rangle P \vee \langle\alpha\rangle Q$ are equally satisfiable. Indeed, satisfiability of the whole is preserved if $\langle\alpha\rangle P$ is replaced by <u>true</u> when P is satisfiable and <u>false</u> when not. Thus we can proceed recursively, working up from the lowest diamonds to determine satisfiability of progessively larger portions of the formulae.

<u>Axioms for programs with loops</u>

For programs with loops we have the following axioms and rules.

$\langle\alpha^n\rangle P \supset \langle\alpha^*\rangle P$ Axioms of Intent (one for each n).

$P \supset [\alpha]P \vdash P \supset [\alpha^*]P$ Rule of Invariance.

$n > z \wedge P \wedge Q(n) \supset \langle\alpha\rangle \exists w (z \leq w < n \wedge Q(w))$

$\vdash n \geq z \wedge Q(n) \supset \langle\alpha^*\rangle (\neg P \wedge \exists w (z \leq w \leq n \wedge Q(n)) \vee Q(z))$

Rule of Convergence

These axioms and rules are explained and justified in more detail in [21]. The first says that if α can halt in some number of steps then α^* can halt. The second says that if α leaves P unchanged, then so does α^*. (Observe how convenient it is to reason about iteration expressed in this form.) The third says that if α "drives" towards z without passing it, provided P remains true, then eventually α^* will either make P false somewhere on the way to z, or it will reach z.

Example proof

The following program was devised by Manna and Pnueli [16] to illustrate the efficacy of their method of proving termination.

```
Y := 0;
     while X ≠ 0 do
       (while X ≠ 0 do (X := X-1; Y := Y+1);
        Y := Y-1;
        while Y ≠ 0 do (Y := Y-1; X := X+1))
```

This program represents an obscure way of setting both X and Y to 0, namely by first setting Y to 0, then copying X into Y by repeatedly decrementing X and incrementing Y, then decrementing Y once, then copying Y back into X by repeatedly decrementing Y and incrementing X. This process is repeated until X becomes 0. The point of this example was that it was supposed to be difficult to prove termination of this program by Floyd's method but easy by the method described by Manna and Pnueli. Our own interest in this program besides the question of ease of proving termination (not a problem in dynamic logic) is that it is just the right size to illustrate the proof techniques appropriate to dynamic logic.

We may write this program in the programming language dynamic logic caters for thus.

```
P1: X>0?; X:=X-1; Y:=Y+1.
P2: Y>0?; Y:=Y-1; X:=X+1.
P3: X>0?; P1*; X=0?; Y:=Y-1; P2*; Y=0?.
P4: Y:=0; P3*; X=0?.
```

P1 represents one step of "copying" a number from X to Y, while P2 represents one step of copying from Y to X. P1*;X=0? and P2*;Y=0? each represent the entire copying process, from X to Y and back again. P3 amounts to a program that, provided Y is initially 0, decrements X. P4 is then the whole program for setting X and Y to 0. The statement we want to prove is, <P4>t (t denotes true), which asserts that it is possible for P4 to halt. The following is the proof, which uses 7 hypotheses from arithmetic and 13 theorems. This proof is machine-readable.

```
% The Manna-Pnueli program %
P1: X>0?; X:=X-1; Y:=Y+1.
P2: Y>0?; Y:=Y-1; X:=X+1.
P3: X>0?; P1*; X=0?; Y:=Y-1; P2*; Y=0?.
```

P4: Y:=0; P3*; X=0?.

% Formulae occurring commonly in the proof%

Ax(n): X=n∧Y=0. Bx(m,n): X=n∧X+Y=m.

Ay(n): Y=n∧X=0. By(m,n): Y=n∧X+Y=m.

% Assumptions from arithmetic - not proved here %

H1: Z=n+1 ≡ Z-1=n. H2: W=n+1 ⊃ W>0.

H3: Ax(n) ≡ Bx(n,n). H4: Ay(n) ≡ By(n,n).

H5: Ax(n) ≡ By(n,0). H6: Ay(n) ≡ Bx(n,0).

H7: W=W.

Thm1: Bx(m,n+1) ⊃ <P1>Bx(m,n).

Show Thm1 {p∧r ⊃ <s?>(p∧r)} using H2 {p⊃s}.

Thm2: By(m,n+1) ⊃ <P2>By(m,n).

Show Thm2 {p∧r ⊃ <s?>(p∧r)} using H2 {p⊃s}.

Thm3: Bx(m,n) ⊃ <P1*>Bx(m,0).

Use Convergence(n) Thm3 from Thm1.

Thm4: By(m,n) ⊃ <P2*>By(m,0).

Use Convergence(n) Thm4 from Thm2.

Thm5: Ax(n) ⊃ <P1*>Ay(n).

Show Thm5 {p⊃<a>q}

    using Thm3 {p⊃<a>q}.

Thm6: Ax(n+1) ⊃ <P1*>Ay(n+1).

Show Thm6 {p} using Thm3 {p}.

Thm7: Ay(n) ⊃ <P2*>Ax(n).

Show Thm7 {r⊃<a>s}

    using Thm4 {p⊃<a>q}, H4 {r≡p}, H5 {s≡q}.

Thm8: Ay(n+1) ⊃ <Y:=Y-1>Ay(n).

Show Thm8 {p∧r ⊃ q∧r} using H1 {p≡q}.

```
Thm9: Ax(n+1) ⊃ <P3>Ax(n).
Show Thm9 {xn1∧y0 ⊃ <xp?;a;x0?;dy;b;y0?>(xn∧y0)}
    using H2 {xn1⊃xp},
                Thm6 {xn1∧y0 ⊃ <a>(yn1∧x0)},
                Thm7 {yn∧x0 ⊃ <b>(xn∧y0)},
                Thm8 {yn1∧x0 ⊃ <dy>(yn∧x0)}.

Thm10: Ax(n) ⊃ <P3*>Ax(0).
Use Performance(n) Thm10 from Thm9.

Thm11: X=n⊃<Y:=0>(X=n∧Y=0).
Show Thm11 {p⊃p∧q} using H7 {q}.

Thm12: X=n⊃<P4>t.
Show Thm12 {p⊃<a;c;r?>t}
    using Thm10 {p∧q⊃<c>(r∧q)}, Thm11 {p⊃<a>(p∧q)}.

Thm13: <P4>t.
Show Thm13 {p} using Thm12 {q⊃p}, H7 {q}.
```

To avoid being distracted by extraneous issues such as arithmetic truth we have introduced all arithmetic facts in this proof as assumptions. (In fact, in the implemented system we have a very fast proof-checker for quantifier-free Presburger arithmetic, using quasi-Gaussian elimination.)

The above proof is not the largest proof we have successfully checked with our system. A substantial part of a total correctness proof of the Knuth-Morris-Pratt pattern-matching algorithm has been machine-checked, and we are in the process of completing this proof. This extends work on the partial correctness of this algorithm by Wegbreit [24].

Discussion of the proof-checker

We have constructed a system for checking proofs of the kind exemplified above. In this we are following in the footsteps of Milner [20,21,26], who is doing for Scott's Logic of Computable Functions what we are doing for the above modal extension to first-order logic. Inasmuch as we are treating programs that manipulate their environments, we are also continuing a tradition of several years of implementing systems for proving and checking proofs of properties of programs [4,8,13,14,23,24]. However the greater expressive power of dynamic logic compared to that of

partial correctness assertions (the language used in almost all such systems) adds considerably to the interest of our system. This consideration actually makes Milner's system a closer relative of ours than the partial correctness systems, due to the greater emphasis on "expressions as first class citizens" in Milner's system and ours, resulting in a logic where programs and facts mingle more freely than say with Hoare's notation. The major difference between Milner's system and ours is the LCF treatment of programs (computable functions) as individuals in the underlying domain versus our treatment of programs as "adverbs," analogously to quantifiers. Another system related to ours is Richard Weyhrauch's [1,25] FOL (First-Order Logic) proof-checker. A detail in which our program differs from Milner's and Weyhrauch's (apart from the obvious one of choice of logical language) is that our program makes less of an effort to help the user interactively than is done by either LCF or FOL, but rather is, at least thus far, a system in which the user prepares his proof exactly as though he were writing a program. This means that his proof exists on a file and is read by the proof-checker just as an interpreter reads a program from a file. This has permitted us to focus all of our effort on the proof-checker proper.

The proof-checker is implemented on the PDP-10 computer at M.I.T.'s Artificial Intelligence Laboratory. The program written to date has aproximately 100 LISP functions comprising a total of 1800 lines of code averaging 4 LISP atoms per line. The bulk of this code is for formula manipulation. However, a small amount of it is for book-keeping tasks of a relatively minor nature associated with keeping track of the structure of a proof.

Directions for further research

Although our immediate goals may not appear to be particularly ambitious or difficult to achieve, as well as not being obviously "Artificial Intelligence" research, we admit to far more ambitious and less plausible objectives on a larger time scale. Ultimately we see the proof-checker itself becoming a component of a variety of very intelligent program-manipulating programs. This depends on our belief that the ability to check proofs is a vital part of any program that pretends to "understand" some domain of discourse where the discussion is at all involved. Two applications that we would like to explore when the proof-checker has reached a satisfactory level of performance are (i) the automatic production of reliable software and (ii) machine-mediated reasoning about programs. Our plan of attack for each of these areas is not presently so crisp that we would feel confident embarking on either area forthwith, particularly the second, but we can nevertheless at this early stage present thoughts on these subjects.

The notion of program reliability through correctness proofs has gained momentum in the past few years, spurred on most notably by the axiomatic methods of Floyd [7] and Hoare [11]. As yet there is not a shred of hard evidence to suggest that this approach supplies the most

economical approach to reliability (where the economics takes into account both the cost of having unreliable software and the total programming and maintenance cost). Indeed, it may well turn out that the bulk of the problems encountered today with unreliable software may be disposed of by a happy combination of a good programming language and a clean programming style. Nevertheless, if the proof-oriented approach can be made to work and does not put too great a burden on the programmer and/or the computer, it may provide reliable software at low cost. We feel it is well worth continuing the experiments that have been going on in this area in the past few years. Although these experiments have not thus far demonstrated the value of correctness proofs, it is still too early to draw any negative conclusions about the method in general.

From a longer-range viewpoint, the burden of programming should become progressively more the computer's responsibility, requiring the computer to "understand" better the programs it executes. This has been the trend since the first assembler was used, and though the trend is perhaps not as pronounced as some have hoped, there is no doubt that the trend continues. As it does, methods of reasoning about programs will concomitantly become a more essential part of the computer's repertoire. This raises the question of the choice of language most appropriate to such reasoning. In view of the expressive power of dynamic logic we feel that it is worth developing the methodology of reasoning in this language with an eye to automating the reasoning as far as possible. A program like our proof-checker is precisely what is needed in the way of a "black box" that "accepts" a reasonably sized step in a discussion about a program. The sort of machine-mediated discussions we envisage could quite well be cast as proofs, albeit in the form of a dialogue. If the notion of a dialogue as a proof seems strange, visualize a conversation - about a program - punctuated with "I don't see why you need that test there" and "How do you guarantee that X will never become negative?" Such conversations about programs arise all the time, and it is clear that the questions are referring to proofs, probably expressed informally but proofs nonetheless. One might argue that proof-checking is not understanding, but we would insist that it is at least a component of understanding.

As humans are taken progressively further out of the loop (admittedly a very long-range view) the dialogue will become more of a monologue. However it may still be appropriate for the computer to reason about the programs it is contemplating using a language like dynamic logic. Thus even in this scenario the basic proof-checking methodology may continue to be used. We should add that we see nothing strange in the idea of a computer checking proofs that it generated itself; the best way to generate proofs may be to propose possibly faulty proofs and subject them to detailed criticism. This would require not only the error-detecting capability of our proposed proof-checker but error-correcting capabilities as well.

Acknowledgments

David Harel and Albert Meyer made substantial contributions to the theoretical underpinnings of dynamic logic. We thank Derek Oppen and Richard Weyhrauch for many helpful ARPA-net-mediated conversations on theorem-proving and proof-checking.

Bibliography

[1] Aiello, M. and R. W. Weyhrauch, Checking Proofs in the Metamathematics of First Order Logic. Computer Science Dept, Stanford University, August, 1974. (50 pages).

[2] Ashcroft, E. and Z. Manna. The translation of 'go to' programs to 'while' programs. STAN-CS-71-188, Stanford, CA. 1971.

[3] de Bakker, J.W., and D. Scott. An outline of a theory of programs. Unpublished manuscript, 1969.

[4] Deutsch, L.P. An Interactive Program Verifier. Ph. D. Thesis, Dept. of Computer Sci., U.C. Berkeley, 1973.

[5] Dijkstra, E. A Discipline of Programming. Prentice-Hall, Englewood Cliffs, N.J. 1976.

[6] Fischer, M.J. and R.E. Ladner. Propositional Modal Logic of Programs. Proc. 9th Ann. ACM Symp. on Theory of Computing, 286-294, Boulder, Col., May 1977.

[7] Floyd, R.W. Assigning Meanings to Programs. In Mathematical Aspects of Computer Science (ed. J.T. Schwartz), 19-32, 1967.

[8] Good, D.I., R.L. London and W.W. Bledsoe. "An Interactive Program Verification System." IEEE Trans. Software Eng., SE-1, 1, 59-67. March 1975.

[9] Harel, D., A.R. Meyer and V.R. Pratt. Computability and Completeness in Logics of Programs. Proc. 9th Ann. ACM (SIGACT) Symposium on Theory of Computing, Boulder, CO, 1977.

[10] Hitchcock, P. and D. Park. Induction Rules and Termination Proofs. In Automata, Languages and Programming (ed. Nivat, M.), IRIA. North-Holland, 1973.

[11] Hoare, C.A.R. An Axiomatic Basis for Computer Programming. CACM 12, 576-580, 1969.

[12] Hughes, G.E. and M.J. Cresswell. An Introduction to Modal Logic. London: Methuen and Co Ltd. 1972.

[13] Joyner, W. H., C. B. Leeman, and W. C. Carter, "Automated Verification of Microprograms", IBM RC5971, April, 1976. (30 pages)

[14] King, J.C. A program verifier. Proc. IFIP Cong. 71, North-Holland, Amsterdam, 1971, 235-249.

[15] Kripke, S. Semantical considerations on Modal Logic. Acta Philosophica Fennica, 83-94, 1963.

[16] Manna, Z. and A. Pnueli. Axiomatic Approach to Total Correctness of Programs. Acta Informatica, 3, 253-263, 1974.

[17] -----, and R. Waldinger. Is "sometime" sometimes better than "always"? Intermittent assertions in proving program correctness. Proc. 2nd Int. Conf. on Software Engineering, Oct. 1976.

[18] Mendelson, E. Introduction to Mathematical Logic. Van Nostrand, N.Y. 1964.

[19] Milner, R.G. Implementation and Applications of Scott's Logic for Computable Functions. Proc. ACM Conf. on Proving Assertions about Programs, (SIGPLAN Notices, 7, 1; SIGACT News, 14), 1-6. Las Cruces, NM, Jan. 1972.

[20] Milner, R., L. Morris and M. Newey, A Logic for Computable Functions with Reflexive and Polymorphic Types, University of Edinburgh, LCF Report No. 1, January, 1975. (25 pages).

[21] Pratt, V.R. Semantical Considerations on Floyd-Hoare Logic. Proc. 17th Ann. IEEE Symp. on Foundations of Comp. Sci., 109-121. 1976.

[22] Robinson, J.A. A Machine-oriented Logic Based on the Resolution Principle. J. ACM 12, 1, 23-41. Jan. 1965.

[23] Suzuki, N., "Automatic Verification of Programs with Complex Data Structures", Stanford University, AIM-279, February 1976, (198 pages)

[24] Wegbreit, B., Constructive Methods in Program Verification. IEEE Trans. on Software

Engineering, SE-3, 3, 193-209. May 1977.

[25] Weyhrauch, R. W., and A. J. Thomas, FOL: a Proof Checker for First-order Logic. Computer Science Dept, Stanford University, AIM-235, September, 1974. (55 pages).

[26] Weyhrauch, R., and R. Milner, Program Semantics and Correctness in a Mechanized Logic. First USA-Japan Computer Conference, 1972. (9 pages).